A Simple Algorithm of Centralized Flow Management for Data Centers

Andrei E. Tuchin*, Masahiro Sasabe*, and Shoji Kasahara*
*Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0192, Japan
Email: andrei.tuchin.ak7@is.naist.jp, m-sasabe@ieee.org, kasahara@ieee.org

***Abstract*—In this paper, we consider a data-flow management mechanism for data center networks, in which a centralized controller called arbiter manages data flows. We propose a simple algorithm for the arbiter to distribute flows over different time points and paths, in a preemptive scheduling and traffic load-balancing manner. The proposed algorithm is base on table-driven resource reservation, in which states of all the links in a data-center network are registered in a single table, and its information is updated whenever a new flow-request arrives at the arbiter. We evaluate the performance of the proposed algorithm through simulation experiments, investigating bit allocation rate and flow allocation rate, under different flow-size distributions. Numerical results show that the proposed algorithm can achieve high bit allocation rate without collisions. It is also shown that the proposed algorithm can allocate many flows within a small time interval even when the variance of flow-size is large.**

I. Introduction

Recently, cloud computing services are provided by data centers with a huge number of server machines. One of performance issues in data centers is how to avoid network congestion when a massive volume of intermediate processing data are exchanged among servers. Recent studies reveal the statistical characteristics in data-center networks. The authors in [1] reported that 99% of data flows are smaller than 100 MBs and that more than 90% of bytes are in flows between 100 MBs and 1 GB. Also an important point is network buffer queue fluctuation of data center networks and the Internet. [2] reported that there are many impulses in the Internet queue fluctuation and a network congestion can be recognized by using binary quantization. For data centers, the queue length variation tend to be uniform, because flow parameters among flows in data center networks are similar. In this paper we exploit this characteristic and the proposed algorithm shows best performance for less flow attributes variations among flows.

In general, a data file is divided into packets with the same source-destination address, and these packets traverse through network as a flow. Recent data-center networks consist of high-speed full-duplex links such as 10 Gb/s, and the capacity of such links is large enough for accommodating a large amount of data packets.

Contributuion: we consider a centralized system for packet flow control in data centers. Data flows inside a data center are fully controlled by the arbiter. We propose a simple algorithm for the arbiter to distribute flows over different time points and paths, in a preemptive scheduling and traffic load-balancing manner. The algorithm aims to avoid collisions among flows and prevent incast throughput collapse inside the data-center network [3]. To achieve this goal, we consider a table-driven resource reservation mechanism, in which states of all the links in the network are registered in a table, and its information is updated at new flow-request arriving points. With the knowledge of network topology of the data center, the proposed mechanism can efficiently avoid network collision and achieve high bit allocation rate. We evaluate the performance of the proposed algorithm through simulation experiments, investigating bit allocation rate and flow allocation rate, under different flow-size distributions.

The rest of this paper is organized as follows. Section II represents the previous studies related to data-center networking. Section III describes the data-center model considered in the paper, and the proposed algorithm is presented in Section IV. We show some numerical examples in Section V, and finally, we conclude the paper in Section VI.

II. Related Work

In recent years, many ideas were proposed to solve the challenge of incast throughput collapse. According to the classification proposed in the a comparative article [4], our solutions can be classified in conformity with next parameters: traffic load-balancing, a centralized system, requires host modifications and a preemptive scheduling.

In [5], the authors propose a centralized and non TCP-based data flow control mechanism called Fastpass. As reported in [4], there are not exist effective systems, algorithms or frameworks that join load-balancing and preemptive scheduling ideas. Fastpass join these two ideas and showed good numerical results. In Fastpass, an arbiter arranges packet transfer requests in a reservation basis. When a source node sends a packet to a destination node, Fastpass determines a timeslot and the path used for that transmission. The authors in [5] developed an experimental system in the Facebook's data center, investigating the performance of Fastpass. Fastpass does not require switch modifications, sender/receiver Network interface controller (NIC), but requires to modify host OSs. In addition, the arbiter of Fastpass suffers from large overhead in packet-based control. In terms of the architecture, in this paper we extends the main idea of Fastpass, total centralized controlling control of states of data center network, but our system

distinguish the load-balancing and the time slot reservation modules which gives ample opportunities for a development of new features and extensions.

Hedera [6] is a non TCP-based centralized data flow distribution system based on OpenFlow [7], in which a core scheduler can access forwarding tables of all core switches such that data flows are dynamically distributed over data-center network. Hedera can efficiently distribute data flows over the network and requires only switch modifications. However, the system is oriented towards long flows and considers one as the major cause of network congestion. The latency of transmissions is not taken into consideration.

pFabric [8] is a priority-based packet-dropping system to reduce latency of packet transmissions. Each packet has priority assigned by source servers. If the buffer of a switch is full and an incoming packet has lower priority than all buffered packets, it is dropped. The proposed solution requires servers and switches modifications.

Deadline-Driven Delivery (D^3) [9] is a distribution system for meeting flow deadlines. The system requires servers and switches modifications. The source server sends the needed transmission rate for a flow to the corresponding destination server. Switches between source-destination servers receive these rate requests and assign required transmission rates. The throughput rate distributions among flows bases on a greedy strategy. This system, however, does not consider load-balancing and incast problem.

Preemptive Distributed Quick (PDQ) protocol [10] tries to quickly complete flow transfer quickly for meeting flow deadlines. PDQ extends the main idea of D^3, however uses more information and logical steps. PDQ uses a priority mechanism based on scheduling disciplines of earliest deadline first and shortest job first. This is a decentralized system which can solve collisions among flows by using states for flows and order of flows for transmission via bottleneck switches. It is reported that PDQ can save 30% average flow completion time compared with the existing transport protocol such as TCP and RCP. However, the protocol is quite challenging to implement in practice and inherits from D^3 problems.

DC-Vegas [2] is an algorithm to control congestion in data centers. The algorithm belongs to the family of TCP congestion-avoidance in the Internet, but DC-Vegas solves the problem of recognizing network congestion in data centers and updating a congestion control window. The updating process bases on Round Trip Time (RTT) and the minimum observed RTT. Other steps are unchanged from the standard algorithm. Numerical results show that a developed delay-based TCP algorithm effectively controls the congestion in data center networks. DC-Vegas requires sender modifications and does not consider load-balancing.

III. System Model

A. Network Topology

We consider a network topology as shown in Fig. 1. The network configuration of these figures is set according to [11].

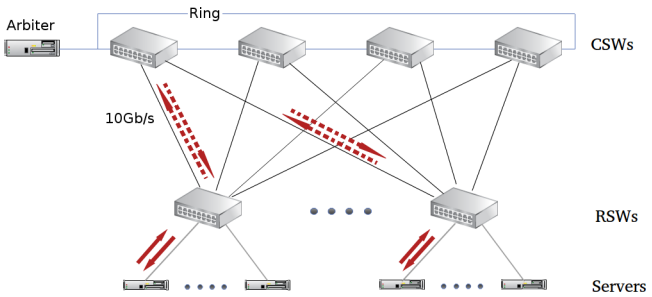

Fig. 1: Intra- and inter-rack flows.

The network considered in the paper consists of the arbiter node, core switches (CSWs), rack switches (RSWs), and server nodes. We assume that CSWs are connected in a ring manner, and that each CSW is connected to all the RSWs, each of which accommodates server nodes.

Let L and M denote the number of CSWs and that of RSWs in the data-center network, respectively. Each CSW has M different links to RSWs. Let C_i $(i = 1, 2, \ldots, L)$ and R_j $(j = 1, 2, \ldots, M)$ denote the CSW with index i and the RSW with index j, respectively.

B. Inter- and Intra-Flows

Flows have the following attributes: size, source server, source rack, destination server, destination rack, time when a flow is generated, time when a flow have to be sent from source server to destination server, and path.

We classify flows into two categories: intra-rack flows and inter-rack ones. An intra-rack flow is a flow whose source and destination servers are in the same rack. An inter-rack flow is a flow whose source and destination servers are located in different racks. In this case, the flow traverses from the source RSW to the destination RSW, via one of CSWs. Inter-rack flows use the resources of CSW. Fig. 1 shows intra- and inter-rack flows.

C. System Architecture

Fig. 2 illustrates main components of the proposed system. The arbiter consists of three components: a server module, a timeslot allocation module and a path selection module. The server module implements functionality for processing requests from servers and interacts with other arbiter's modules. The timeslot allocation module assigns a timeslot for a flow transmission request depend on information in table-driven mechanism for resource reservation, IV-B. The path selection module independently from the timeslot allocation module constraints a structure of table-driven mechanism, IV-A. In general, a load-balancing and preemptive scheduling operates independently, however a final response to a server is constrained based on information from these two modules.

A client side consists of an additional client module to send requests to the arbiter and manipulates a NIC. The client module can be realized by modifying OS kernel without any hardware modifications.

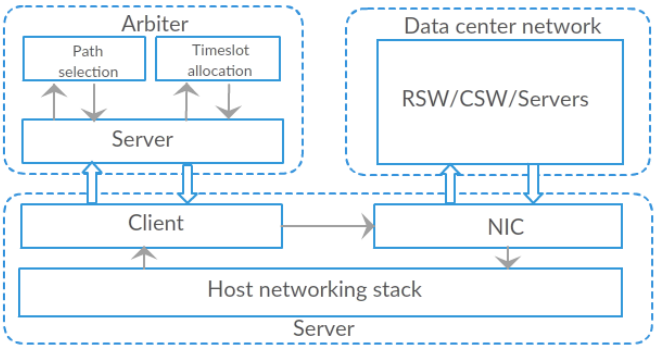

Fig. 2: System architecture.

IV. The Proposed Algorithm

In this section, we present the core algorithm of the arbiter. The arbiter is located at the highest hierarchy of the network topology, where all the CSWs belong to, and hence the arbiter is connected to all the CSWs. The arbiter is the centralized controller for scheduling data flows in the data center network. When a server node s has data to transfer to destination server node d, the server node s sends a flow request to the arbiter before transferring data packets to node d. When the arbiter receives the request from node s, the arbiter schedules the flow from s to d by reserving all the links along the route from s to d.

A. Server Grouping and CSW Mapping

In this paper, the link capacity is regarded as a resource for data flows. The basic idea of the proposed scheme is that the whole capacity of a link is reserved for one flow in the minimum time interval during which all the data packets composing the flow are transmitted. For example, when a request of a flow with 10 Gbits arrives at the arbiter, the arbiter reserves the 10 Gb/s output link of a CSW for one second.

We define $n^{(i)}$ as the number of servers accommodated in RSW R_i ($i = 1, 2, \ldots, M$). The servers accommodated in R_i are classified into L groups. Let $G_k^{(i)}$ ($k = 1, 2, \ldots, L$, $i = 1, 2, \ldots, M$) denote the set of server indices in R_i, each of which indicates the destination server of inter-rack flow coming from the other rack via CSW C_k. Note that

$$\sum_{k=1}^{L} |G_k^{(i)}| = n^{(i)},$$

where $|G_k^{(i)}|$ is the cardinal number of set $G_k^{(i)}$. We define G as the index set of all the servers in the data center, given by $G = \cup_{i=1}^{M} \cup_{k=1}^{L} G_k^{(i)}$.

Note that the server classification plays an important role for achieving load balancing of the proposed algorithm. In this paper, we assume that $G_k^{(i)}$'s are determined a priori for all k and i according to data traffic profiles so that the resulting traffic pattern is well distributed.

B. Core Algorithm

The basic idea is that when the arbiter receives a new flow request, the arbiter reserves the output link(s) of the RSW and CSW along the flow path. More precisely, if the request is for an intra-rack flow, its source and destination servers are located in the same rack, and the arbiter reserves the output link of the rack, which is connected to the destination server. If the request is for an inter-rack flow, on the other hand, it's source and destination servers are placed in different racks, and hence the arbiter reserves the output link from CSW to RSW and that from the RSW to the destination server, along the flow path.

The main issue of the arbiter algorithm is how the information of each flow reservation is managed. To tackle this issue, we introduce two fundamental elements: the link-utilization table and the freshening time. The link-utilization table records which links are used for the packet flows, while the freshening time is the time at which the link-utilization table is cleared, indicating that no link is used for flow transmission.

The outline of the proposed algorithm is as follows. For simplicity, we assume the capacity of all the links in the data center is the same and equal to C. Consider the case in which a source node $s \in G_k^{(i)}$ tries to send a flow of B bits to a destination $d \in G_l^{(j)}$. If $i = j$, destination d is accommodated in the same rack of R_i, and hence the packet flow from s to d is an intra-rack flow with route

$$s \to R_i \to d.$$

If $i \neq j$, on the other hand, destination d is located in the different rack of R_j and the packet flow from s to d is an inter-rack flow with route

$$s \to R_i \to C_l \to R_j \to d.$$

We define $l(n_1, n_2)$ as the link from node n_1 to node n_2. In the above example of inter-rack flow, the flow path from s to d consists of $l(s, R_i)$, $l(R_i, C_l)$, $l(C_l, R_j)$, and $l(R_j, d)$ links.

Let S denote the set of CSWs, RSWs, and server nodes in the data center network, given by

$$S = \{C_1, \ldots, C_L\} \cup \{R_1, \ldots, R_M\} \cup S_{\mathrm{S}},$$

where S_{S} denote the set of server nodes in the data center. We further define $S^{(l)}$ as the set of links in the data center, given by

$$S^{(l)} = \{l(n_1, n_2) : n_1, n_2 \in S\}.$$

Now we define the link-utilization table $LUtable(n_1, n_2)$ for link $l(n_1, n_2) \in S^{(l)}$ as follows.

$$LUtable(n_1, n_2) = \begin{cases} 1, & \text{if link } l(n_1, n_2) \text{ is used,} \\ 0, & \text{otherwise.} \end{cases}$$

We will show the instance of $LUtable$ in Section IV-D.

We define T_{f} as the freshening time at which all flow transfers completed or will complete. When the current time is before T_{f}, the system has ongoing flow(s) in the network. When T_{f} is past, on the other hand, all the flow transfers

complete, and hence all the elements of $LUtable$ can be cleared to zero. We define $T_{\mathrm{f\_prev}}$ as the previous freshening time which was used before T_{f}. $T_{\mathrm{f\_prev}}$ and T_{f} are borders of a time window for transmission of a flow sequence without colliding with each other.

Now suppose that at time t, the arbiter receives a new request of a flow from $s \in G_k^{(i)}$ to $d \in G_l^{(j)}$, whose size is B. Let $S_l(s,d)$ denote the set of links along the path from s to d. $S_l(s,d)$ is given by

$$S_l(s,d) = \begin{cases} \{l(s,R_i), l(R_i,d)\}, & \text{if } i = j, \\ \{l(s,R_i), l(R_i,C_l), l(C_l,R_j), l(R_j,d)\}, & \text{if } i \neq j. \end{cases}$$

With T_{f} and $T_{\mathrm{f\_prev}}$, the proposed algorithm updates $LUtable$ according to the following way.

Case 1: $t > T_{\mathrm{f}}$.

In this case, all links of the data center are not used at time t. The arbiter sets 0's to all the cells of $LUtable$, and then sets

$$LUtable(n_1,n_2) = 1, \quad \forall l(n_1,n_2) \in S_l(s,d).$$

T_{f} and $T_{\mathrm{f\_prev}}$ are updated by

$$T_{\mathrm{f}} := t + B/C, \quad T_{\mathrm{f\_prev}} := t.$$

Note that B/C is the transmission time of the flow with size B.

Case 2: $t \leq T_{\mathrm{f}}$.

In this case, some links of the data center are used at time t. If links that are needed for the flow are not used ($LUtable(n_1,n_2) = 0$, $\forall l(n_1,n_2) \in S_l(s,d)$), the arbiter sets 1's to cells of $LUtable$ as follows:

$$LUtable(n_1,n_2) = 1, \quad \forall l(n_1,n_2) \in S_l(s,d).$$

The freshening time T_{f} is updated by

$$T_{\mathrm{f}} := \begin{cases} T_{\mathrm{f}}, & \text{if } T_{\mathrm{f}} \geq T_{\mathrm{f\_prev}} + B/C, \\ T_{\mathrm{f\_prev}} + B/C, & \text{if } T_{\mathrm{f}} < T_{\mathrm{f\_prev}} + B/C. \end{cases}$$

If some link(s) needed for the flow are used ($LUtable(i,j) = 1$ for some $l(n_1,n_2) \in S_l(s,d)$), the arbiter sets 0's to all the cells of $LUtable$, and then sets

$$LUtable(n_1,n_2) = 1, \quad \forall l(n_1,n_2) \in S(s,d).$$

$T_{\mathrm{f\_prev}}$ and T_{f} are updated by

$$T_{\mathrm{f\_prev}} := T_{\mathrm{f}}, \quad T_{\mathrm{f}} := T_{\mathrm{f}} + B/C.$$

C. Buffering for Colliding Requests

In the algorithm, freshening time T_{f} is updated whenever used links are detected. Consider the case where a link is required for many flow requests at the same time. In this case, T_{f} is updated repeatedly, while most of links are idle and not used for file transfer, resulting in low link utilization. In order to avoid this case, the arbiter is equipped with the buffer of file requests to which the arbiter fails in allocating time intervals. Note that increasing the buffer size also requires much computation, which may result in deteriorating the rate of flow allocation. We will discuss how this buffering affects the performance of the proposed algorithm in Section V.

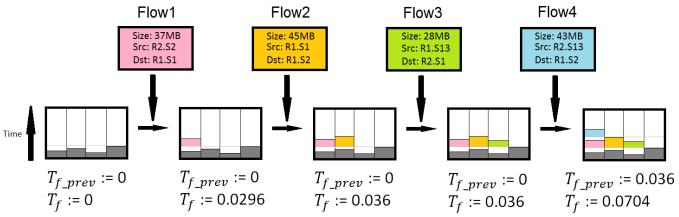

Fig. 3: Example of scheduling.

D. Example of Scheduling

Fig. 3 shows an example of main logical steps for updating $LUtable$, T_{f}, $T_{\mathrm{f\_prev}}$. In this example, we assume the link capacity is 10Gb/s, and that $T_{\mathrm{f}} = 0$, $T_{\mathrm{f\_prev}} \leq 0$. Suppose that the real time is just before 0, and that four flow requests arrive at the arbiter within a very short term. The attributes of the flows are shown in Fig. 3.

Table I shows how $LUtable$ is updated at each flow request. In this table, "-" implies null.

In Table I, $S_k^{(i)}$ is a server node k accommodated in RSW R_i. We assume the following path scenario.

$$\begin{aligned} &\text{Flow 1:} && S_2^{(2)} \to R_2 \to C_1 \to R_1 \to S_1^{(1)} \\ &\text{Flow 2:} && S_1^{(1)} \to R_1 \to S_2^{(1)} \\ &\text{Flow 3:} && S_{13}^{(1)} \to R_1 \to C_2 \to R_2 \to S_1^{(2)} \\ &\text{Flow 4:} && S_{13}^{(2)} \to R_2 \to C_1 \to R_1 \to S_2^{(1)} \end{aligned}$$

Because the first link from a source to a RSW is exclusively available for the source node, we do not need to reserve the first link.

Flow 1 is the flow with 37 MB size, its source server is $S_2^{(2)}$ accommodated in RSW R_2, and the destination is $S_1^{(1)}$ in RSW R_1. Note that Flow 1 is an inter-rack flow. Since the transfer time of Flow 1 is $(37 \times 10^6 \times 8)/(10 \times 10^9) = 0.0296$ s, the arbiter reserves links between its source and destination for the time interval from 0 s to 0.0296 s.

Then, the arbiter reserves links for Flow 2. Flow 2 is an intra-rack flow of R_1, and the destination $S_2^{(1)}$ of Flow 2 is different from the destination of Flow 1, and hence the arbiter can allocate to Flow 2 the time interval starting at 0 for the link between R_1 and $S_2^{(1)}$. Flow 3 is also a good case in the sense that no collision occurs, and hence the arbiter allocates to Flow 3 the time interval starting 0 for the links along the Flow 3 path.

Flow 4 is an inter-rack flow, and its destination is the same as Flow 2. To avoid collision at the link from R_1 to $S_2^{(1)}$, the arbiter updates $T_{\mathrm{f\_prev}}$ to the current value of T_{f}. With the update, the arbiter allocates time intervals starting from $T_{\mathrm{f\_prev}}$.

V. Simulation Results

A. Experimental Setting

In order to evaluate the performance of the proposed algorithm, we conducted simulation experiments. In this simulation, we consider the network topology where the number of

TABLE I: Example of $LUtable$ updating.

$State$	n_1 \ n_2	C_1	R_1	R_2	$S_1^{(1)}$	$S_2^{(1)}$	$S_1^{(2)}$
0	C_1	-	0	0	-	-	-
	R_1	0	-	-	0	0	-
	R_2	0	-	-	-	-	0
1	C_1	-	1	0	-	-	-
	R_1	0	-	-	1	0	-
	R_2	1	-	-	-	-	0
2	C_1	-	1	0	-	-	-
	R_1	0	-	-	1	1	-
	R_2	1	-	-	-	-	0
3	C_1	-	1	1	-	-	-
	R_1	1	-	-	1	1	-
	R_2	1	-	-	-	-	1
4	C_1	-	1	0	-	-	-
	R_1	0	-	-	0	1	-
	R_2	1	-	-	-	-	0

CSWs is 4 and the number of RSWs is $M = 32$. The number of server nodes accommodated in a RSW is identical for all the RSWs and equal to 48. Therefore, the total number of servers in the system is 1536. In each rack, 48 servers are classified into 4 server groups, i.e., $L = 4$ and $G_k^{(i)} = 12$ for all k and i. One CPU core manages one group, i.e., $32 \times 12 = 384$ servers. All the links in the network are full-duplex. The link capacity for a pair of a CSW and a RSW and that for a pair of a RSW and a server node are 10 Gb/s. The capacity of links with which CSWs are connected in a ring fashion is 80 Gb/s. We set buffer size Q to be 100 and 700. Note that the buffer size is the maximum number of flow requests that can be buffered in the arbiter.

In this paper, we mainly focus on the stress test of proposed algorithm, that is, how many flow requests the arbiter can process per second. For this purpose, we generate flow requests as many as possible so that the arbiter always has flow requests to be served. For each flow request, the pair of source server and destination server is randomly chosen from all the servers. The base size of a flow is 12 Kbits (1500 bytes), and the additional data size is generated according to a probability distribution. We consider three distributions for the additional data size: exponential, Weibull, and Pareto distributions. We prepared 50 patterns of flow requests for each case.

The simulation results are compared with Fastpass, Hedera, and an optimal case. In the optimal case, only intra-flow requests are generated such that all the 384 servers in one group can simultaneously transfer data to their destination nodes at the maximum link speed of 10 Gb/s during the entire simulation. Thus, the optimal case can fully utilize the data center network.

As for the performance measure, we consider ideal bit allocation rate, actual bit allocation rate, and actual flow allocation rate. The ideal (resp. actual) bit allocation rate is calculated by dividing the overall transferred data by simulation (resp. execution) time. Note that the simulation time is constant and defined by the scenario, but the execusion time varies depending on multiple factors, e.g., buffer size Q of proposed algorithm and processing power of arbiter. Thus, the ideal bit allocation rate becomes the uppoer bound of actual bit allocation rate. We can also calculate the upper bound of ideal bit allocation rate, which is equal to that for the optimal case, as follows. In the optimal case, all the 384 servers in one group can simultaneously transfer data to its destination node at 10 Gb/s, thus 3.84 Tb/s is achieved. The actual flow allocation rate is calculated by dividing the number of allocated flows by execution time.

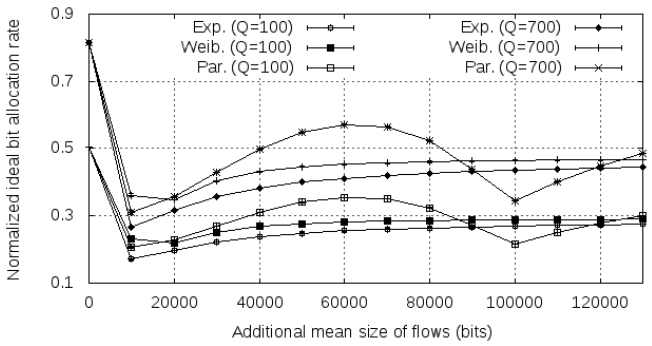

Fig. 4: Normalized ideal bit allocation rate vs. mean additional data size.

The simulation program was implemented with C language, and was performed on 2.5 GHz CPU with 4 GB RAM, running on Ubuntu 14.04. In what follows, we show the mean and standard deviation for each performance measure.

B. Ideal Bit Allocation Rate

Fig. 4 illustrates the relationship between mean additional data size and ideal bit allocation rate for $Q = 100$ and 700. Note that the ideal bit allocation rate is normalized by its upper bound, i.e., 3.84 Tb/s. In each buffer-size case, we show the results for three distributions of additional data size, i.e., exponential, Weibull, and Pareto.

We first observe that the ideal bit allocation rate takes the maximum value at additional data size of zero for all the cases. This is because each flow size is constant and hence it is easy for the arbiter to allocate time intervals to fixed-size flows. When the mean additional data size becomes greater than zero, the ideal bit allocation rate steeply decreases but gradually increases with the additional data size for exponential and Weibull distribution cases. On the contrary, we observe a cyclic tendency in the Pareto distribution case.

Next, we focus on the impact of buffer size Q. Since large buffer can avoid frequent updating of T_f, the results for $Q = 700$ can outperform those of $Q = 100$. Recall that the ideal bit allocation rate is not affected by the computation overhead, which depends on Q. We evaluate the impact of computation overhead in Section V-C.

C. Actual Bit Allocation Rate

In Section V-B, we revealed the ideal characteristics of proposed algorithm, where the computation overhead is ignored. In actual system, however, conducting the proposed algorithm requires a certain time, which depends on multiple factors, e.g., buffer size Q and processing power of the arbiter.

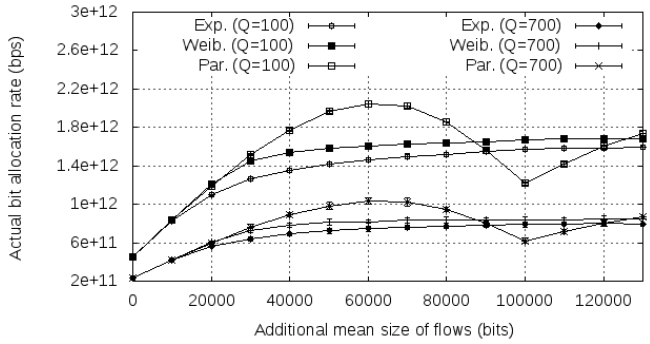

Fig. 5: Actual bit allocation rate vs. mean additional data size.

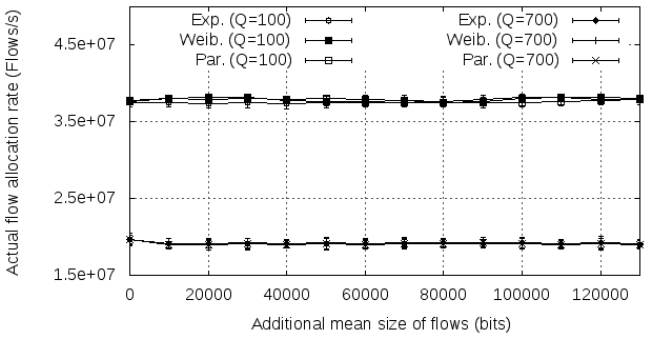

Fig. 6: Actual flow allocation rate vs. mean additional data size.

Fig. 5 illustrates the relationship between mean additional data size and actual bit allocation rate for $Q = 100$ and 700. In each buffer-size case, we show the results for three additional data size distributions.

We observe that the results are similar to those in Fig. 4, for each distribution case.

However, we also find two different characteristics between Fig. 4 and Fig. 5. First, the actual bit allocation rate for $Q = 100$ becomes greater than that for $Q = 700$, regardless of the additional data size distribution. This is because the actual bit allocation rate is affected by the execution time, which increases with buffer size Q. Second, when the mean additional data size is zero, the actual bit allocation rate is small but the ideal bit allocation rate is large. This is also because of much computation time, due to a large number of arriving flow requests. In our stress testing environments, the number of arrival flows increases when the mean additional data size is fixed and small.

Finally, we try to compare the results of proposed algorithm with those of Fastpass. The scenario with 120 Kbits of additional data size is almost equivalent to the experiment condition of Fastpass [5]. It is reported in [5] that Fastpass achieves the bit allocation rate of 0.4 Tb/s, which is obtained at experiments with 4 CPU cores. On the contrary, the proposed algorithm achieves 0.8 Tb/s by a moderate PC with one CPU core, even for $Q = 700$.

D. Actual Flow Allocation Rate

Fig. 6 illustrates the relationship between mean additional data size and actual flow allocation rate for $Q = 100$ and 700. In this figure, the actual flow allocation rate for three distribution cases are almost the same and remain constant in each buffer-size case. This result suggests that the actual flow allocation rate is insensitive to the additional data-size distribution. We also observe that the actual flow allocation rate for $Q = 100$ is greater than that for $Q = 700$, as in Fig. 5.

It is reported in [6] that Hedera can manage about 706 flows per 1 ms for a data-center network with 1024 hosts. In our experiment, the number of server nodes is 1536, however, Fig. 6 shows that the proposed algorithm can allocate more than 18,600 flows per 1 ms, even for $Q = 700$.

VI. Conclusion

In this paper, we developed the algorithm to allocate time interval and path for each data flow in the data center network. The proposed algorithm is based on the table-driven resource reservation mechanism, in which states of all the links in the network are registered in a table, and its information is updated at new flow-request arriving points. Numerical results showed that the proposed algorithm can effectively allocate a large number of flows to link resources, confirming that the proposed algorithm achieves better performance than the existing algorithms.

References

[1] A. Greenberg, J.R. Hamilton and N. Jain, "VL2: A Scalable and Flexible Data Center Network," *ACM SIGCOMM 2009 conference on Data communication*, vol. 39, no. 4, Oct. 2009.

[2] J. Wang, J. Wen and C. Li, "DC-Vegas: A Delay-Based TCP Congestion Control Algorithm for Datacenter Applications," *Journal of Network and Computer Applications*, vol. 53, Jul. 2015.

[3] Y.=Chen, R. Griffith, J. Liu, R. H. Katz and A. D. Joseph, "Understanding TCP Incast Throughput Collapse in Datacenter Networks," *ACM WREN 2009*, pp. 73–82 Oct. 2009.

[4] R. Rojas-Cessa, Y. Kaymak and Z. Dong, "Schemes for Fast Transmission of Flows in Data Center Networks," *IEEE Communications Surveys and Tutorials*, vol. 17, no. 3, Aug. 2015.

[5] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah and H. Fugal, "Fastpass: A Centralized "Zero-Queue" Datacenter Network," *ACM SIGCOMM'14*, Aug. 2014.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," *NSDI*, 2010.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[8] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar and S. Shenker, "pFabric: Minimal Near-Optimal Datacenter Transport," *ACM SIGCOMM'13*, vol. 43, pp. 435–446, 2013.

[9] C. Wilson, H. Ballani and T. Karagiannis, "Better Never Than Late: Meeting Deadlines in Datacenter Networks," *SIGCOMM Comput. Commun.*, vol. 41, no. 4, Aug. 2011.

[10] C. Hong, M. Caesar and P. Godfrey, "Finishing Flows Quickly with Preemptive Scheduling," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 127–138, 2012.

[11] N. Farrington and A. Andreyev, "Facebook's Data Center Network Architecture", *IEEE Optical Interconnects Conf.*, 2013.